

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy**, as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

1. **[Why not just do divide-and-conquer?] (7 points)** Recall the Longest-Common-Subsequence problem from class: given sequences X and Y of length m and n respectively, find the length of a longest common subsequence (LCS) of X and Y . In class we saw a bottom-up dynamic programming approach; in this program will explore a top-down approach.

For parts (a) and (b), refer to the pseudo-code below, which is a divide-and-conquer approach that returns the length of the longest common subsequence of two sequences X and Y :

```
LCS(X,Y):
  i=length(X)
  j=length(Y)
  if i==0 or j==0:
    return 0
  else:
    if X[i]==Y[j]:
      return 1 + LCS(X[1..i-1],Y[1..j-1])
    else:
      return max(LCS(X[1..i],Y[1..j-1]),LCS(X[1..i-1],Y[1..j]))
```

- (a) (3 points) Suppose that X and Y are sequences over an alphabet of size at least 2. Prove that the worst-case runtime of the above recursive implementation of the algorithm is $\Omega(2^{\min(m,n)})$, where m and n are the lengths of the inputs X and Y .

[We are expecting: A short but formal proof. Make sure you specify what the worst-case input that you consider is.]

- (b) (4 points) In class, we saw a bottom-up dynamic program for Longest-Common-Subsequence. Give pseudocode for a top-down version of this algorithm. That is, modify the divide-and-conquer algorithm above to keep a table of solutions to avoid repeated work. Your dynamic programming solution should have running time $O(mn)$, and your algorithm should have the same recursive structure as $\text{LCS}(X,Y)$ above.

[We are expecting: the pseudo-code and a short informal justification of why it runs in time $O(mn)$.]

SOLUTION:

- (a) In order to exhibit worst-case running time, we come up with a worst-case input. Since the alphabet size is at least two, suppose without loss of generality that A and B are in the alphabet, and consider the inputs $X = AAAAAA \dots A$ (m times) and $Y = BBB \dots B$ (n times). Suppose each sub-problem is labelled by (i, j) (notice that the same pair i, j may appear several times). Consider the recursion tree formed by these sub-problems. Notice that whenever $i > 0$ and $j > 0$, a node (i, j) will have two children in this tree: $(i-1, j)$ and $(j-1, i)$. This is because we always have $X[i] = A$ and $Y[j] = B$, so $X[i] \neq Y[j]$. Thus, for the first $\min(n, m)$ levels of the tree (from

the root down), the recursion tree is a full binary tree, and level $\min(m, n)$ has $2^{\min(m, n)}$ nodes in it. Because there is at least a constant amount of work done at each node, the running time of this pseudocode is $\Omega(2^{\min(m, n)})$.

(b) Here is our pseudo-code:

```
LCS_outer loop(X,Y)  \\inputs X and Y
    m=length(X), n=length(Y)
    global variable M (m by n table initialized to be 0's)
    global variable checkDone (m by n binary table, initialized to be 0's)
    return LCS(X,Y)

LCS(X,Y)
    i=length(X), j=length(Y)
    if checkDone[i,j]=1
        return M[i,j]
    else
        if i=0 or j=0
            M[i,j]=0
            checkDone[i,j]=1
            return M[i,j]
        else
            if X(i)=Y(j)
                M[i,j]= 1 + LCS(X[1..i-1],Y[1..j-1])
                checkDone[i,j]=1
                return M[i,j]
            else
                M[i,j]= max(LCS(X[1..j],Y[1..j-1]),LCS(X[1..i-1],Y[1..j]))
                checkDone[i,j]=1
                return M[i,j]
```

2. [Longest path.] (7 points) We saw in class that finding longest simple paths in general graphs doesn't seem to be amenable to dynamic programming. However, in a DAG, it turns out it is. In this question, you'll design a dynamic programming algorithm that finds longest simple paths in a directed acyclic graph.

For the following, let $G = (V, E)$ be a weighted directed **acyclic** graph, so that the weight on an edge $(u, v) \in E$ is $w(u, v)$. Further suppose that $s, t \in V$, and that all vertices in V are reachable from s . Recall that a path is *simple* if it has no cycles. We say that a *longest simple path* from s to t is a path from s to t with maximal cost.

- (a) (2 points) For a vertex $x \in V$, let $L(x)$ denote the cost of a longest simple path from s to x . Show that for all $x \neq s$,

$$L(x) = \max_{u: (u, x) \in E} (L(u) + w(u, x)).$$

[We are expecting: a formal proof. Make sure that you explicitly use the fact that G is acyclic; otherwise the statement is false!]

- (b) (1 point) Suppose that $v_1, \dots, v_n$ is a topological sorting of the vertices. That is, if $(v_i, v_j) \in E$, then $i < j$. Show that
- i. $s = v_1$, and that
 - ii.

$$L(v_j) = \max_{i: i < j \text{ and } (v_i, v_j) \in E} (L(v_i) + w(v_i, v_j)).$$

[We are expecting: a short but formal proof of these two statements. (No more than a sentence or two for each).]

- (c) (4 points) Design a dynamic programming algorithm to find the *length* of the longest directed path from s to t , which runs in time $O(n + m)$, where $n = |V|$ and $m = |E|$.

Notes: If it is helpful, you may assume that you have access to a method `topologicalSort(G)` which, in time $O(n + m)$, takes a graph G with n vertices and m edges, and returns an array `topoSort` so that `topoSort[i]` contains a pointer to the i 'th vertex in a topologically sorted order.¹ As usual, you may also assume that there are methods `getIncomingNeighbors(v)` and `getOutgoingNeighbors(v)` which takes a vertex v and in time $O(1)$ returns a pointer to the head of a list incoming or outgoing neighbors of v , respectively.

[We are expecting: pseudocode, with a short informal explanation of why it is correct and why the running time is $O(n + m)$.]

SOLUTION:

- (a) Since all vertices are reachable from s , there is some longest simple path from s to x , suppose that it is $s, u_1, \dots, u_k, x$. Notice that it may be that $u_k = s$. We claim that the path $P = s, u_1, \dots, u_k$ is a longest simple path from s to u_k . First, since P is a simple path from s to u_k , the cost of P is at most the cost of a longest simple path. Second, the cost of P is at least the cost of a longest simple path; to see this, suppose that there were a path P' from s to u_k that were longer. Then $P' \circ x$ (here, $\circ$ denotes concatenation) is a simple path from s to x , and longer than our original path, a contradiction. Here, we are using the fact that G is a DAG to assert that $P' \circ x$ is a simple path; in a DAG, all paths are simple. Together, this implies that P must be a longest simple path from s to u_k .

Thus, the cost of P is $L(u_k)$, by definition, and so we have

$$L(x) = (L(u_k) + w(u_k, x) \leq \max_u L(u) + w(u, x)$$

On the other hand, since for any u so that $(u, x) \in E$, the longest path from s to u , followed by x , is a path from s to x , and hence

$$L(u) + w(u, x) \leq L(x).$$

Together these two inequalities prove the claim.

- (b) Since every vertex is reachable from s , and G is a DAG, we must have $s = v_i$ and $i < j$ for all $j \neq i$. Thus, $i = 1$.

The second statement is true because we have only added the condition that $i < j$ in the maximum to our result in part (a); but if $(v_i, v_j) \in E$, then $i < j$ anyway.

- (c) `longestSimplePath(G,t):`

```

Initialize an array W[1..n]
Use DFS to topologically sort the vertices in G.
For each vertex, maintain a field labeled "index"
that allows you to get the index of the
vertex in the topological sort in O(1) time,
if you already have the vertex.
// suppose this ordering is v_1, ..., v_n,
// and that t = v_r.
// Since there is a path from s to all other vertices, v_1 = s.
L[1] = 0
// this is the correct base case because the longest simple path from
```

¹Not required: how would you implement such an algorithm?

```

        // s to s has cost 0.
    for i in 2,...r:
        L[i] = 0
        for j so that (v_j, v_i) is in E:
            L[i] = max{ L[i], L[j] + w(v_j, v_i) }
            // because we are iterating through the vertices in topological order,
            // we know that j < i, and so L[j] has already been filled in.
    return L[r]

```

This is correct because we are implementing the recursive relationship between sub-problems that we found in part (a). Notice that because we go through the vertices in topologically sorted order, we never query $L[j]$ before we have filled it out.

The running time is $O(n + m)$. This is because it takes time $O(n + m)$ to do DFS, and then because we iterate over at most m edges between our two for loops.

3. **[Dynamic programming for making change.] (8 points)** Coins in the United States are minted with denominations of $\{1, 5, 10, 25, 50\}$ cents. There are several ways to make change of any given amount of money: for example, to make change of 11 cents, we could use a dime and a penny, two nickels and a penny, or 11 pennies.

In the mythical country of CS161-land, coins are minted with denominations of $\{d_1, \dots, d_t\}$ CS161-cents, where $d_1, \dots, d_t$ are positive integers. In this problem, we will design dynamic programming algorithms to understand how to make change of n CS161-cents.

- (a) (1 point) Suppose that the denominations are $\{1, 6, 10\}$. What is the way to make change of 18 CS161-cents that involves the fewest coins? How many different ways are there to make change of 18 CS161-cents?

[We are expecting: just the answers to these questions]

- (b) (3 points) Suppose that $M[k]$ is the minimum number of coins needed to represent k CS161-cents. Give an efficient dynamic programming algorithm which maintains a table M , and which takes an integer n and a set D of denominations as input, and outputs a **way to make change** of n CS161-cents using the minimal number of coins. (Not just the number of coins needed).

[We are expecting: pseudocode, and an informal justification that it is correct.]

- (c) (1 point) Analyze the running time of your algorithm in part (b).

[We are expecting: The best big-oh running time you can guarantee, and a short justification.]

[We are expecting: pseudocode, and an informal justification that it is correct.]

- (d) (3 points) Suppose that, for $j \leq t$, $N[j, k]$ is number of ways to make change of k CS161-cents using only the denominations $\{d_1, \dots, d_j\}$. Give an efficient dynamic programming algorithm which maintains a table N , and which takes an integer n and a set D of denominations as input, and outputs the number of **distinct** ways to make change of n CS161-cents using the denominations in D .

Assume that coins of the same denomination are indistinguishable. For example, if $D = \{1, 6, 10\}$ and $n = 6$, then your algorithm should return 2, because there are two ways to make change of 6: 6 itself and 1, 1, 1, 1, 1, 1.

[We are expecting: pseudocode, and an informal justification that it is correct.]

- (e) (1 point) Analyze the running time of your algorithm in part (e).

[We are expecting: The best big-oh running time you can guarantee, and a short justification.]

SOLUTION:

(a) There are six ways to make change of 18 CS161-cents:

- i. 18 1's
- ii. 10, 6, 1, 1
- iii. 10 and eight 1's
- iv. 6 and twelve 1's
- v. Two 6's and six 1's
- vi. Three 6's.

The one that uses the fewest number of coins is the last, which uses three 6-CS161-cent coins.

(b) `minChange(n,D):`
 Initialize a table `M[0..n]`
 Initialize a table `R[0..n]`
 `M[0] = 0`, set `M[k] = Infinity` for all `k != 0`
 `R[0] = [0 for i in range(t)]`
 // the way to make change of 0 cents is to have 0 of each coin.
 for `k = 1, ..., n`:
 `j* = None`
 for `j = 1, ..., t` so that `d_j <= k`:
 `M[k] = min{ M[k], M[k - d_j] + 1 }`
 if `M[k]` was updated:
 `j* = j`
 `R[k] = R[k - d_j*].copy()` // this takes time $O(t)$ to copy over
 `R[k][j*] += 1`
 return `R[n]`

In the pseudocode above, we keep an array `R`, so that $R[k]$ is the best way to make change out of k items, in addition to the suggested array `M`. If we find that the best thing to do is to remove d_j and use the optimal solution for making change of $k - d_j$ CS161-cents, we will take the previous best solution and add d_j to it.

This works because in the inner loop over i , we are setting

$$M[k] = \min_j \{M[k - d_j] + 1\},$$

Using the interpretation of M given in the problem, we see that this is indeed a valid recurrence relation for the minimum number of coins needed to make change of k CS161-cents. This is because if the optimal solution involves denomination d_j , then the solution that leaves out one coin of value d_j is an optimal solution for making change of $k - d_j$.

(c) The running time of this algorithm is $O(nt)$. The outer loop is over t things. The inner loop goes over t things and does $O(1)$ work (accessing an array) inside. The other work done inside the outer loop is copying over `R[k]`, which also takes time $O(t)$.

(d) `numWays(n,D):`
 Initialize a t -by- n table `N[0..t,0..n]`.
 `N[0,k] = 0` for all `k = 1, ..., n`
 `N[i,0] = 1` for all `i = 1, ..., t`
 `N[0,0] = 1`
 for `k = 1, ..., n`:
 for `j = 1, ..., t`:
 `N[j,k] = N[j-1,k]`
 if `D[j] <= k`:
 `N[j,k] += N[j,k-D[j]]`

return N[t,n]

The above pseudo-code uses the recurrence relation

$$N[j, k] = N[j - 1, k] + N[j, k - d_j].$$

This is correct because the number of ways to make change out of k CS161-cents using only the denominations $d_1, \dots, d_j$ is equal to the number of ways to make change without using d_j , or by using at least one copy of d_j .

- (e) The running time is $O(nt)$. This is because the outer loop runs for n times, the inner loop runs for t times, and there is constant work per iteration because all the previous entries have already been calculated when filling in a new entry.
4. **[Fish fish eat eat fish.] (6 points)** Plucky the Pedantic Penguin enjoys fish, and he has discovered that on some days the fish supply is better in Lake A and some days the fish supply is better in Lake B. He has access to two tables A and B , where $A[i]$ is the number of fish he can catch in Lake A on day i , and $B[i]$ is the number of fish he can catch in Lake B on day i , for $i = 1, \dots, n$. Assume that $A[i]$ and $B[i]$ are positive integers for all $i = 1, \dots, n$.

Plucky's goal is to have as many fish as possible at the end of day n . If he happens to be at Lake A on day i and wants to be at Lake B on day $i + 1$, he may pay L fish to a polar bear who can take him from Lake A to Lake B overnight; the same is true if he wants to go from Lake B back to Lake A. Here $L > 0$ is a positive integer. Assume that before day 1 begins, Plucky is at Lake A, and he has zero fish.

Plucky will save all the fish he gets until the end of day n (at which point he will feast), but the polar bear does not accept credit. So Plucky must have the fish on hand in order to pay the polar bear; he must pay *before* he travels.

In this question, you will design an $O(n)$ -time dynamic programming algorithm that finds the maximum number of fish that Plucky will have on day n . Do this by answering the two parts below.

- (a) (3 points) What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between a problem and the sub-problems? What is the base case for this recursive relationship? Justify your answer.

[We are expecting: a formal definition of your sub-problems, as well as a recursive relationship that they satisfy. We are also expecting an informal justification that your recursive relationship is correct given your definitions.]

- (b) (3 points) Write pseudocode for a dynamic programming algorithm that takes as input A, B, L and n , and in time $O(n)$ returns the maximum number of fish that Plucky can eat on day n .

[We are expecting: the pseudocode, a brief explanation (one or two sentences) about why it works, and justification (also one or two sentences) that it runs in time $O(n)$.]

SOLUTION:

- (a) Our sub-problems will be indexed by a flag ℓ (either A or B), an integer $i \in \{1, \dots, n\}$. Let $K[\ell, i]$ denote the number of fish that Plucky owns on day i , assuming he is in location ℓ on that day. The recursive relationship is:

$$K[A, i] = \begin{cases} \max\{K[A, i - 1] + A[i], K[B, i - 1] - L + A[i]\} & K[B, i - 1] \geq L \\ K[A, i - 1] + A[i] & \text{else} \end{cases}$$

and

$$K[B, i] = \begin{cases} \max\{K[B, i-1] + B[i], K[A, i-1] - L + B[i]\} & K[A, i-1] \geq L \\ K[B, i-1] + B[i] & \text{else} \end{cases}$$

The base case is that $K[A, 1] = A[1]$ and $K[B, 1] = -\infty$.

To see that this is correct, consider just the first one. The maximum amount of fish that Plucky can eat if he is in location A on day i is the maximum of two possibilities: either he was in location A on day $i-1$, stayed in location A , and gained $A[i]$ fish; or else he was in location B on day $i-1$, paid L to the polar bear to move from B to A , and then ate $A[i]$ fish. Plucky only has the possibility of traveling if his current amount of fish, $K[A, i-1]$, exceeds L . This explains the expressions in the first equation. The second equation is similar.

(b) `fishFishEatEatFish(A, B, L, n):`
 initialize a 2-by- n array K so that the first coordinate is indexed by $\{A, B\}$
 $K[A, 1] = A[1]$
 $K[B, 1] = -\text{Infinity}$
 for $i = 2, \dots, n$:
 $K[A, i] = K[A, i-1] + A[i]$
 if $K[B, i-1] \geq L$:
 $K[A, i] = \max\{K[A, i], K[B, i-1] - L + A[i]\}$
 $K[B, i] = K[B, i-1] + B[i]$
 if $K[A, i-1] \geq L$:
 $K[B, i] = \max\{K[B, i], K[A, i-1] - L + B[i]\}$
 return $\max\{K[B, n], K[A, n]\}$

This works because it is implementing the recursive relationship from part (a). The base case is that on day 1, we have $K[A, 1] = A[1]$ (Plucky can stay and fish) and $K[B, 1] = -\infty$ (Plucky is not allowed to start Day 1 at Lake B , since he has no fish to pay the polar bear before day 1 begins).

The running time is $O(n)$, because the loop is over $i = 2, \dots, n$, and the amount of work inside each iteration is $O(1)$.

5. **[There is no problem 5.] (0 points)** This problem set is long enough already! But if you finish all of the above and still want more practice with dynamic programming, try the problems in Chapter 15 of CLRS.